Documentation of `ModelSolver` [*]
A `Python` class for analyzing dynamic algebraic models

Magnus Kvåle Helliesen [†]

April 12, 2023

Abstract

This paper documents the `Python` class `ModelSolver`. `ModelSolver` lets the user define a model object in terms of equations and endogenous variables. It contains methods to solve the model subject to data (in a `Pandas DataFrame`), as well as analyzing the model using graph theory and network plots.

What sets `ModelSolver` apart from other similar solvers, is that it does not require the equations of the model to be written in any particular form, or that the user associates equations with endogenous variables. Most other solvers require that either 1) the model is normalized (i.e., that the model is written in terms of endogenous variables), or 2) that the user explicitly associates equations with endogenous variables. This is non-trivial for models with lots of equations. `ModelSolver`, however, reads equations in whatever form they may be formulated, and performs the necessary analyzes, without any input from the user other than lists of equations and endogenous variables.

`ModelSolver` was developed in order to facilitate solving an input-output model for the Norwegian monthly national accounts. It analyzes and solves the model's more than 15,500 equations over more than 30 periods in under a minute on a laptop computer.

[*] https://github.com/statisticsnorway/model-solver.git
[†] mkh@ssb.no

Contents

1 Background **2**
1.1 The model . . . 2
1.2 Block analysis and ordering . . . 3
1.3 Simulation code . . . 5
1.4 Solution . . . 6

2 Examples of use **7**
2.1 Solving the model . . . 7
2.2 Analyzing the model . . . 7

3 Dependencies **10**

A Proofs **10**

1 Background

1.1 The model

Consider a model consisting of n equations, [1]

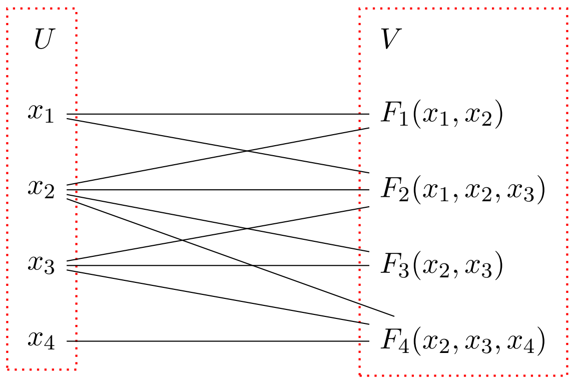
$$\begin{aligned} L_1(\mathbf{x}_t|\mathbf{z}_t) &= R_1(\mathbf{x}_t|\mathbf{z}_t), \\ L_2(\mathbf{x}_t|\mathbf{z}_t) &= R_2(\mathbf{x}_t|\mathbf{z}_t), \\ &\vdots \\ L_n(\mathbf{x}_t|\mathbf{z}_t) &= R_n(\mathbf{x}_t|\mathbf{z}_t), \end{aligned}$$

for $t = T_0, T_0 + 1, \ldots, T_1$, where $\mathbf{x}_t = (x_{1,t}, x_{2,t}, \ldots, x_{n,t})$ is a vector of endogenous variables (and, the way it is written, the values of which solve the model), and $\mathbf{z}_t = (z_{1,t-k_1}, z_{2,t-k_2}, \ldots, z_{m,t-k_m}, \ldots, x_{i,t-\kappa_i}, \ldots)$, for $k_i \geq 0$ and $\kappa_i > 0$, is a vector of contemporaneous and lags of exogenous variables and lags of endogenous variables (or taken together: *predetermined* variables). [2] It's convenient to formulate the model using a vector function $\mathbf{F} : \mathbb{R}^n \mapsto \mathbb{R}^n$, [3]

$$\mathbf{F}(\mathbf{x}_t|\mathbf{z}_t) = \begin{pmatrix} F_1(\mathbf{x}_t|\mathbf{z}_t) \\ F_2(\mathbf{x}_t|\mathbf{z}_t) \\ \vdots \\ F_n(\mathbf{x}_t|\mathbf{z}_t) \end{pmatrix} = \begin{pmatrix} L_1(\mathbf{x}_t|\mathbf{z}_t) - R_1(\mathbf{x}_t|\mathbf{z}_t) \\ L_2(\mathbf{x}_t|\mathbf{z}_t) - R_2(\mathbf{x}_t|\mathbf{z}_t) \\ \vdots \\ L_n(\mathbf{x}_t|\mathbf{z}_t) - R_n(\mathbf{x}_t|\mathbf{z}_t) \end{pmatrix},$$

[1] L_i and R_i denote the left and right hand side of equation i, respectively.
[2] Note that $\mathbf{z}_t$ may contain only $z_{i,t-k_i}$'s or $x_{i,t-k_i}$'s or be empty.
[3] We say that $\mathbf{F}$ maps from $\mathbb{R}^n$ because exogenous variables (contemporaneous and lags) and lags of the endogenous variables are taken as given in any given period, i.e. predetermined.

Figure 1: Bipartite graph (BiGraph of model)



with the solution to the model being given by $\{\mathbf{x}_t\}_{t=T_0}^{T_1}$ such that

$$\mathbf{F}(\mathbf{x}_t|\mathbf{z}_t) = \mathbf{0} \text{ for } t = T_0, T_0 + 1, \ldots, T_1.$$

On the face of it, this is a problem which the *Newton-Raphson*-algorithm handles well. The issue, however, is that n might be quite large. In the Norwegian national accounts, for instance, n is greater than 15,500. Therefore, it is useful to analyze the system of equations before solving it, in order to break it down into minimal *simultaneous blocks* that can be solved in a particular order.

1.2 Block analysis and ordering

In order to analyze and divide the model into blocks, we use results from *graph theory*.

First, we construct a *bipartite graph* (BiGraph) that connects endogenous variables with equations. This is illustrated in Figure 1 for an arbitrary model with four equations (we omit time subscripts and exogenous variables and lags for notational convenience).

Next, we apply *maximum bipartite matching* (MBM), which assigns each endogenous variable to *one and only one* equation. The MBM is arbitrary, and Figure 2 shows one possible solution.

We use the MBM to determine what endogenous variables impact what *other* endogenous variables. We do this by 1) assigning endogenous variables to equations (or ‘labeling’ the equations ‘as’ endogenous variables) using the MBM (the MBM, after all, is a one-to-one mapping between endogenous variables and equations), and 2) removing the MBM edges (the MBM edges simply imply that each endogenous variable affects itself, which, although it does not alter the result, is non-informative). The causality goes from U to V. This is illustrated in Figure 3.

Figure 2: Maximum Bipartite Match (MBM) of BiGraph

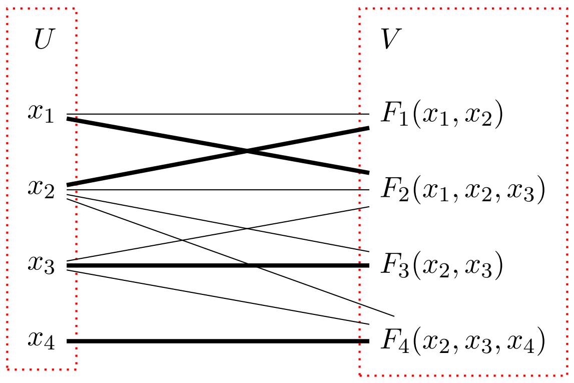
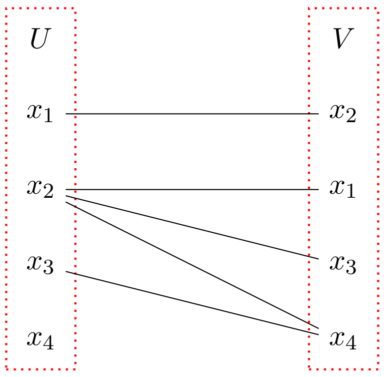


Figure 3: Graph of what endogenous variables impact what *other* endogenous variables



Although the MBM is arbitrary, the description of what endogenous variables affect what other is not (see proof).

Next, we use Figure 3 to construct a *directed graph* (DiGraph) that shows how the endogenous variables impact each other. The DiGraph is shown in Figure 4.

Finally, we find the *strong components* of the DiGraph, as shown in Figure 5. A strong component is a set of nodes that are connected such that every node can be reached from every other node in the set (traversing the arrows). A *condensation* of the DiGraph is a (new) DiGraph with each node being a strong component (of the former). The condensation can be illustrated as in Figure 6. Each node of the condensation corresponds to a block of the model, and the arrows decide the order in which the blocks are to be solved. See [1] for a discussion of strong components and block analysis.

Note that, since the MBM is a one-to-one-mapping, we may draw the

Figure 4: Directed graph (DiGraph) of what endogenous variables impact what *other* endogenous variables

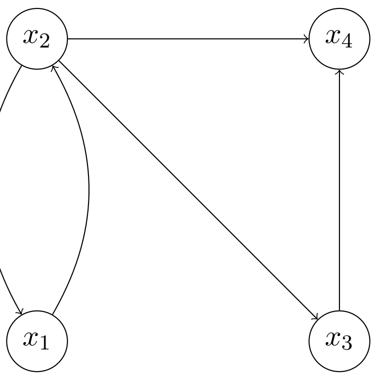
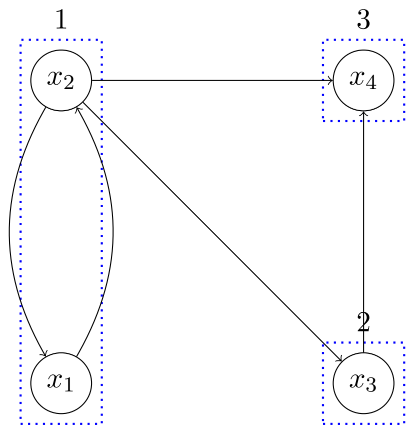


Figure 5: Strong components of DiGraph



DiGraph and condensation labeling the nodes with endogenous variables or equations.

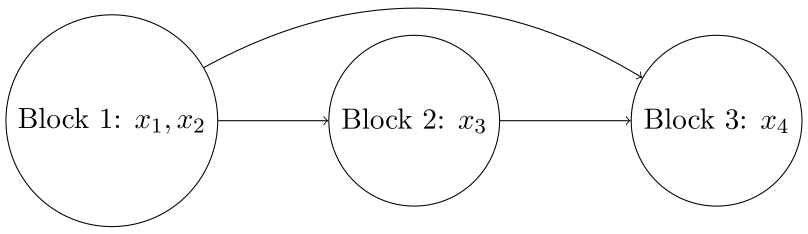
In this example, x_1 and x_2 must be solved first in one simultaneous block. Next, x_3 is solved (taking x_1 and x_2 as given, being predetermined by the solution to block 1). Finally, x_4 is solved (taking x_1, x_2 and x_3 as given, being predetermined by the solutions to block 1 and 2).

1.3 Simulation code

With the blocks of the model as given by the condensation of the model DiGraph, as previously discussed, we can generate *simulation code* for each block. That is *either*

- a symbolic function (definition) that takes predetermined input and

Figure 6: Condensed DiGraph



returns the endogenous value (if the block is a definition, as described below), *or*

- a symbolic *objective function* and *Jacobian matrix* that take predetermined input and (initial) values for the endogenous variables, which are to be sent to a Newton-Raphson algorithm, which in turn returns the endogenous value(s)

A block is a said to be a definition *if and only if* it 1) consists of one equation and 2) the only thing on the left hand side of the equation is the endogenous variable of that equation (and that endogenous variable does *not* show up on the right hand side of that equation too).

1.4 Solution

In order to solve the model numerically, we loop (nested) over time periods (chronologically) and blocks (according to the order dictated by the condensation of the model DiGraph).

If the block to be solved is a definition, as discussed above, the solution is given straight forwardly by the function defining it (given predetermined input).

If the block to be solved is not a definition, it is sent to a Newton-Raphson algorithm, which in turn returns the solution (given predetermined input, and initial values for the endogenous variables). The $k+1$st iteration of the Newton-Raphson algorithm is given by

$$\mathbf{x}_t^{(k+1)} = \mathbf{x}_t^{(k)} - \mathbf{J}_{\mathbf{F}}^{-1}(\mathbf{x}_t^{(k)}|\mathbf{z}_t)\mathbf{F}(\mathbf{x}_t^{(k)}|\mathbf{z}_t),$$

where $\mathbf{J}_{\mathbf{F}}(\mathbf{x}_t|\mathbf{z}_t)$ is the Jacobian matrix of $\mathbf{F}(\mathbf{x}_t|\mathbf{z}_t)$ evaluated at $\mathbf{x}_t$ and $\mathbf{z}_t$. The algorithm stops and returns $\mathbf{x}_t^{(k+1)}$ if $\max\left(\left|\mathbf{x}_t^{(k+1)} - \mathbf{x}_t^{(k)}\right|\right) \leq \varepsilon$, where ε is a tolerance level. If it so happens that $\mathbf{F}(\mathbf{x}_t^{(0)}|\mathbf{z}_t) = \mathbf{0}$, then the initial values solve the model, and as such $\mathbf{x}_t^{(0)}$ is returned as the solution.

2 Examples of use

The `ModelSolver` object is instantiated by

```
model = ModelSolver(equations, endogenous)
```

where `equations` and `endogenous` are lists containing equations and endogenous variables as strings, e.g.,

```
equations = [
'x1 = a1',
'x2 = a2',
'0.2*x1+0.7*x2 = 0.1*ca+0.8*cb+0.3*i1',
'0.8*x1+0.3*x2 = 0.9*ca+0.2*cb+0.1*i2',
'k1 = k1(-1)+i1',
'k2 = k2(-1)+i2'
]

endogenous = ['x1', 'x2', 'ca', 'cb', 'k1', 'k2']
```

Note that the `(-#)`-syntax denotes lags of a variable; `ModelSolver` does not support leads. Mathematical functions supported are `max()`, `min()`, `log()` and `exp()`. Note also that the equations are written in a form that is neither normalized, nor associates equations with endogenous variables.

It is possible to switch endogenous and exogenous variables using

```
model.switcR_endo_vars(old_endo_vars, new_endo_vars)
```

where `old_endo_vars` and `new_endo_vars` are lists. Upon invoking `switcR_endo_vars`, a tuple of endogenous variables stored inside the model object is updated, and `ModelSolver` immediately performs block analysis on the updated model, such that it can be solved for the updated list of endogenous variables.

2.1 Solving the model

Let `input_df` be a `Pandas DataFrame` containing exogenous variables and initial values for the endogenous variables. Then

```
solution_df = model.solve(input_df)
```

solves the model and stores the results in `solution_df`, which is a `Pandas DataFrame` identical to `input_df`, *but* where the endogenous variables are solutions to the model.

2.2 Analyzing the model

`ModelSolver` contains a number of methods that lets the user analyze the model.

`describe()` describes the model in terms of number of equations and blocks, along with information about the blocks, e.g.,

```
Model consists of 6 equations in 5 blocks
4 of the blocks consist of simple definitions

4 blocks have 1 equations
1 blocks have 2 equations
```

`find_endo_var(<endogenous variable>)` returns the block in which the endogenous variable is solved for.

`show_block(<block>)` returns information about the block, i.e., endogenous variables, exogenous and predetermined variables and equations. E.g.,

```
Block consists of an equation or a system of equations

2 endogenous variables:
cb ca

4 exogenous or predetermined variables:
i2 i1 x1 x2

2 equations:
0.8*x1+0.3*x2 = 0.9*ca+0.2*cb+0.1*i2
0.2*x1+0.7*x2 = 0.1*ca+0.8*cb+0.3*i1
```

`show_blocks()` returns information about all the blocks of the model.

`trace_to_exog_vars(<block>)` traces the model DiGraph back to the nodes of origin and reports what exogenous variables determine those nodes.

`trace_to_exog_vals(<block>, <period index>)` traces the model DiGraph back to the nodes of origin, finds what exogenous variables determine those nodes, and reports values for those variables for the chosen period index (0,1,...).

`draw_blockwise_graph()` draws a network graph as shown in Figure 7. The method takes as arguments a variable name, and the maximum number of ancestor and descendant nodes, and maximum number of nodes in total.

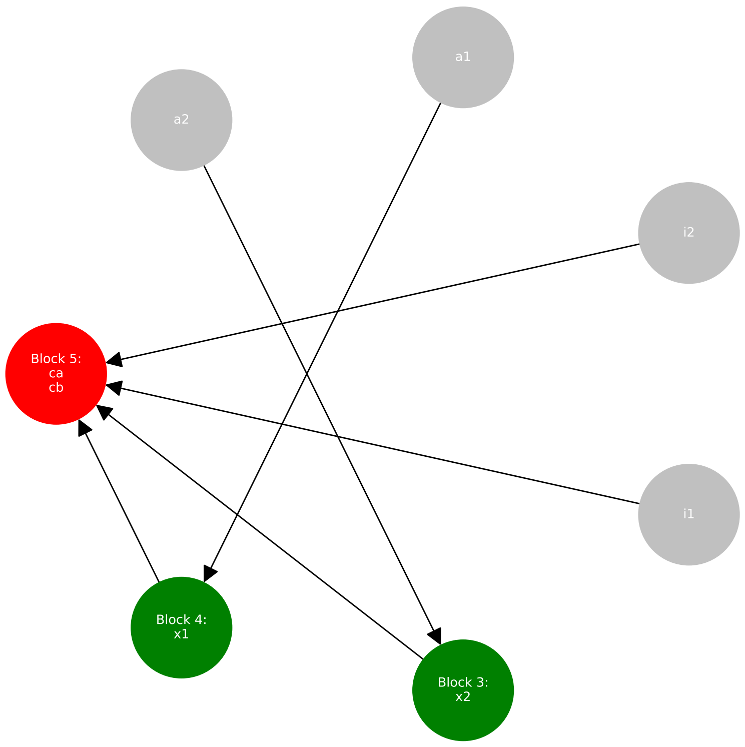


Figure 7: Example of network plot

3 Dependencies

`ModelSolver` is built using the following packages:

- NumPy
- NetworkX
- Pandas
- SymEngine
- Numba
- collections
- functools
- Matplotlib

References

[1] Gilli, Manfred. "Causal Ordering and Beyond." International Economic Review, vol. 33, no. 4, 1992, pp. 957–71. JSTOR, https://doi.org/10.2307/2527152. Accessed 22 Mar. 2023.

A Proofs

Theorem 1. *All MBMs, $M_1, M_2, \ldots$, result in the same description of what endogenous variables impact what* other *endogenous variables, $E'_1 = E'_2 = \ldots = E'$.*

Proof. Suppose we have a BiGraph $G(U, V, E)$, with

$$E = \{(x_i, F_p), (x_i, F_q), (x_j, F_p), (x_j, F_q)\} \cup \tilde{E},$$

where $\tilde{E}$ is a set containing any *other* edges between endogenous variables and equations (or functions) F than the ones explicitly given. Suppose further that G has two MBMs (G may have more MBMs, in fact as many as $n!$, but we only care about these two for the purpose of this proof), [4]

$$M_1 = \{(x_i, F_p), (x_j, F_q)\} \cup \tilde{M},$$
$$M_2 = \{(x_i, F_q), (x_j, F_p)\} \cup \tilde{M},$$

[4] As we can choose x_i, x_j, F_p and F_q, this is always the case, unless the MBM is unique (which would render the proof superfluous).

that is, M_1 and M_2 are the same, except for two edges (the ones matching x_i and x_j with F_p and F_q; $\tilde{M}$ is a set containing all *other* matches). We use the MBMs to label equations with endogenous variables, as discussed above, i.e.,

$$\begin{aligned} E_1' &= \{\cancel{(x_i, x_i)}, (x_i, x_j), (x_j, x_i), \cancel{(x_j, x_j)}\} \cup \tilde{E}', \\ E_2' &= \{(x_i, x_j), \cancel{(x_i, x_i)}, \cancel{(x_j, x_j)}, (x_j, x_i)\} \cup \tilde{E}'. \end{aligned}$$

where prime denotes that we have labeled equations as endogenous variables using the MBM. (We ignore the mappings from an endogenous variable onto itself.) Note that $\tilde{E}'$ is the same for E_1' and E_2'. This follows because we have chosen $\tilde{M}$ to be the same for M_1 and M_2. Therefore, we see that $E_1' = E_2'$. Since we can go from any MBM to any other switching two and two edges repeatedly, it follows that *any* MBM gives the same result as *any other* E', which is what we set out to prove. $\square$